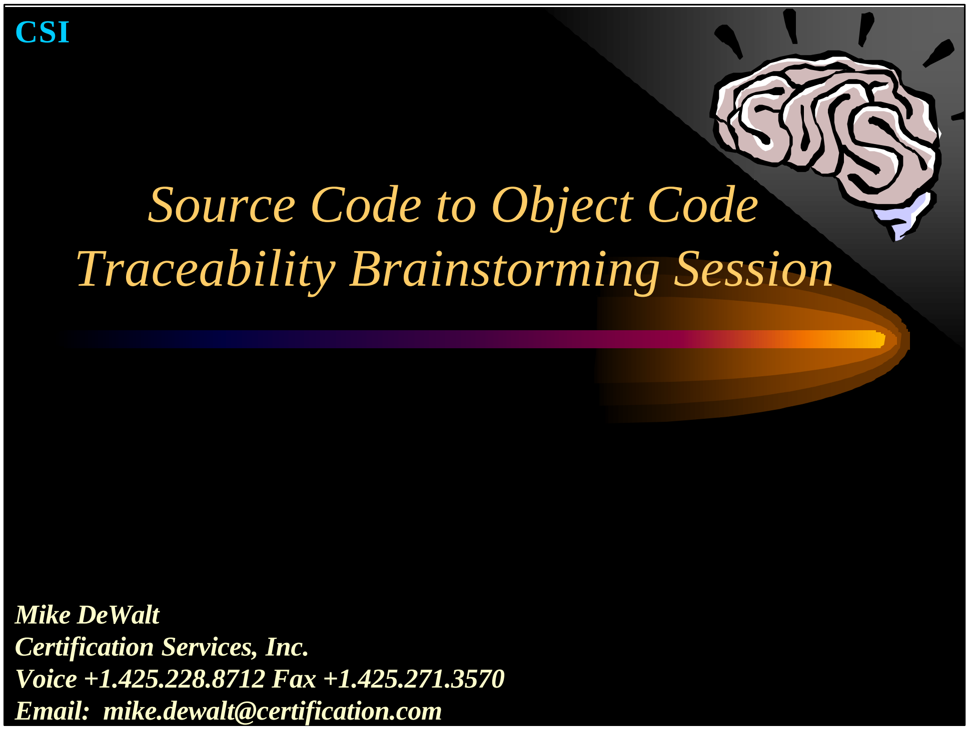
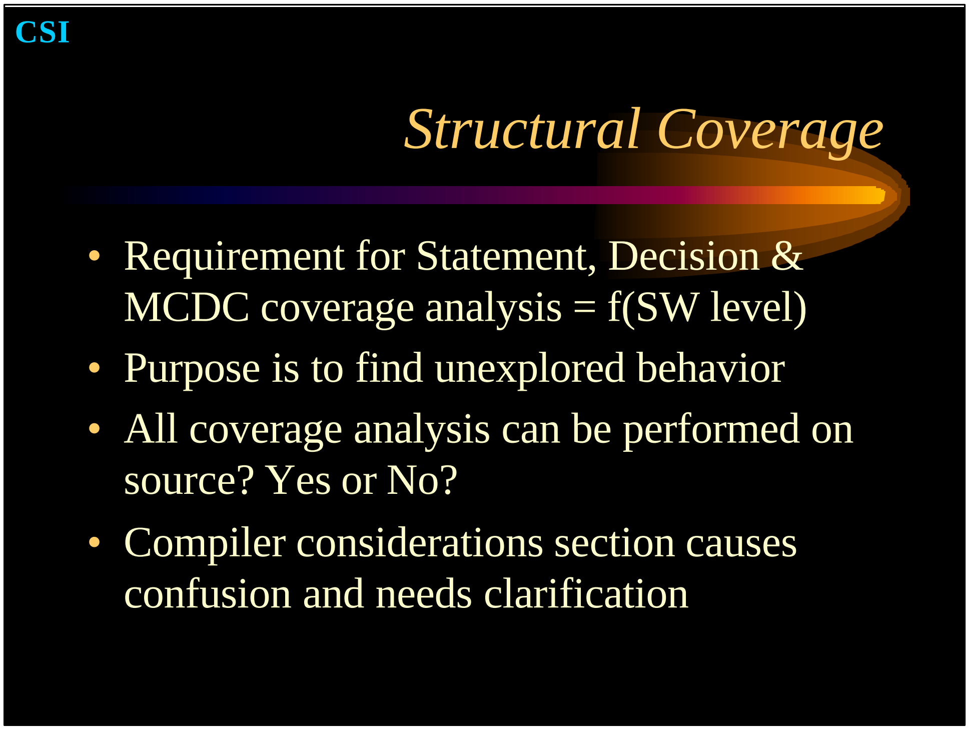


CSI

Source Code to Object Code Traceability Brainstorming Session

Mike DeWalt
Certification Services, Inc.
Voice +1.425.228.8712 Fax +1.425.271.3570
Email: mike.dewalt@certification.com

This brainstorming session is designed to produce ideas and potential text for a notice on source to object code traceability. This is mainly concerned with the interpretation of DO-178B/ED-12B sections 4.4.2 (b) and 6.4.4.2(b). A grass-horse version of the notice is provided as a starting point.

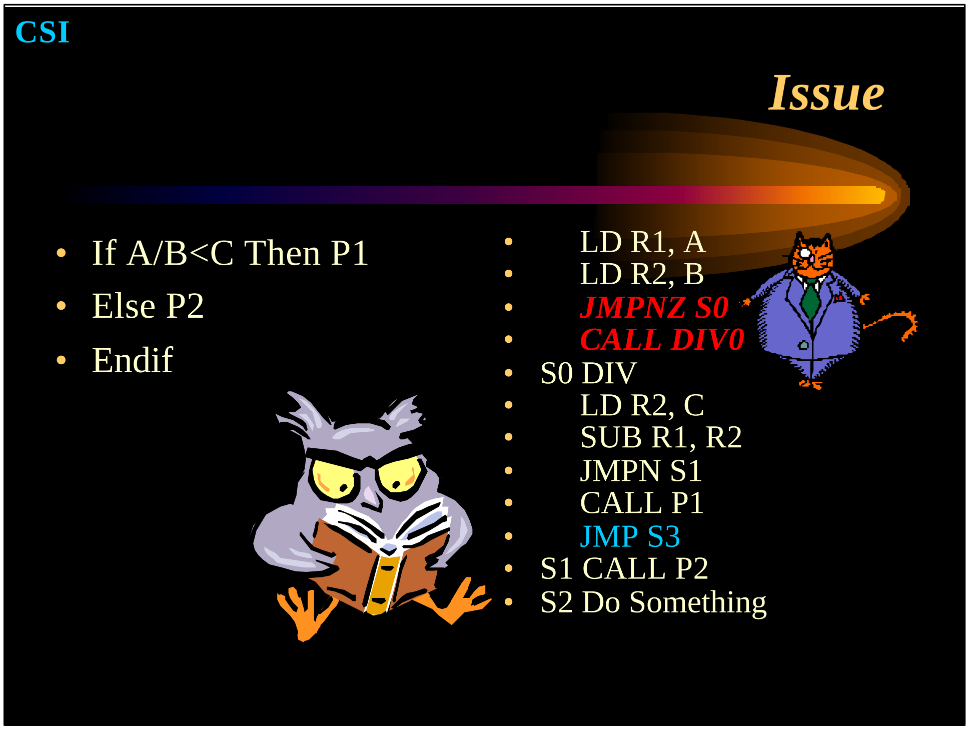


CSI

Structural Coverage

- Requirement for Statement, Decision & MCDC coverage analysis = f(SW level)
- Purpose is to find unexplored behavior
- All coverage analysis can be performed on source? Yes or No?
- Compiler considerations section causes confusion and needs clarification

Test coverage analysis addressed in section 6.4.4 of DO-178B is embodied in a number of the objectives of the document. The two components of this analysis are Requirements-Based Test Coverage Analysis in section 6.4.4.1 and Structural coverage Analysis in section 6.4.4.2. The objectives for structural coverage analysis only apply for software levels A, B, and C . The purpose of this analysis is to determine which code structure was not exercised by the requirements based tests. Section 6.4.4.2 (b) indicates that this structural analysis may be performed on the source code. For level A software, DO-178B section 6.4.4.2 (b) indicates that additional verification may be required if the compiler generates object code that is not directly traceable to the Source Code Exactly what additional verification is needed and how source can be shown directly traceable to object code has been confusing to regulatory authorities and industry and has resulted in inconsistent application of this section. Section 4.4.2 (b), Language and Compiler Considerations, provide some insight into what is meant by traceability issue. As can be established from the above extract from DO-178B, the purpose of the traceability exercise is to identify added functionality not visible at the source code level. Specific examples were initializations, built in error handling, etc. Therefore any traceability exercise need only demonstrate that the exercise is capable of detecting this type of added functionality. Some people have read this to mean disassembly of the object into the source and others have assumed this requires MCDC at the object code.



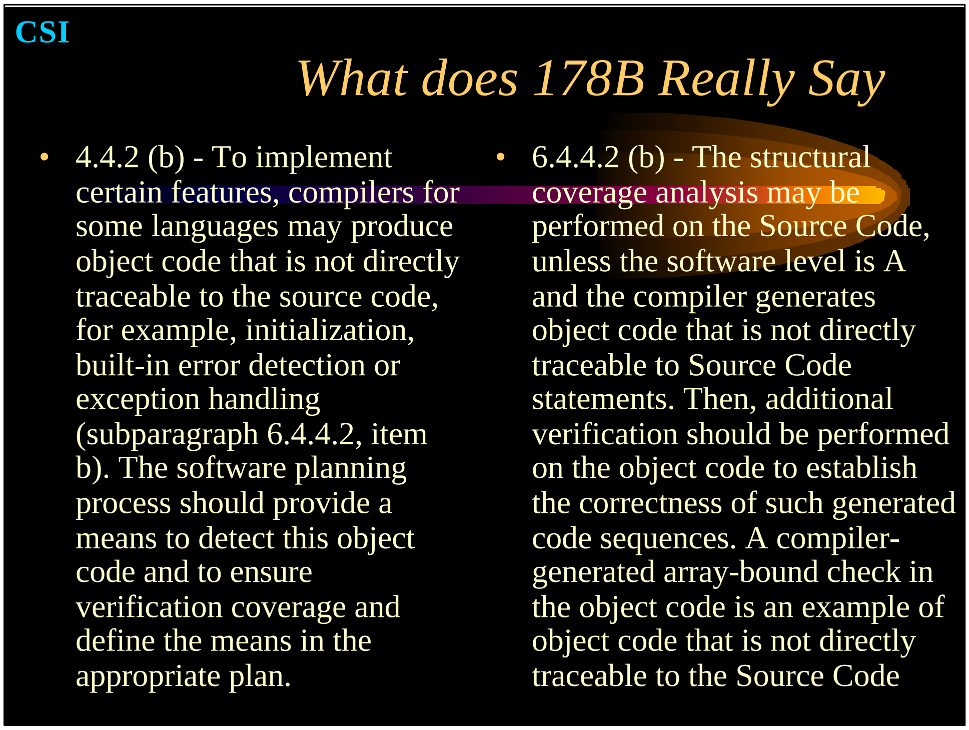
CSI

Issue

- If A/B<C Then P1
- Else P2
- Endif

```
•     LD R1, A
•     LD R2, B
•     JMPNZ S0
•     CALL DIV0
• S0 DIV
•     LD R2, C
•     SUB R1, R2
•     JMPN S1
•     CALL P1
•     JMP S3
• S1 CALL P2
• S2 Do Something
```

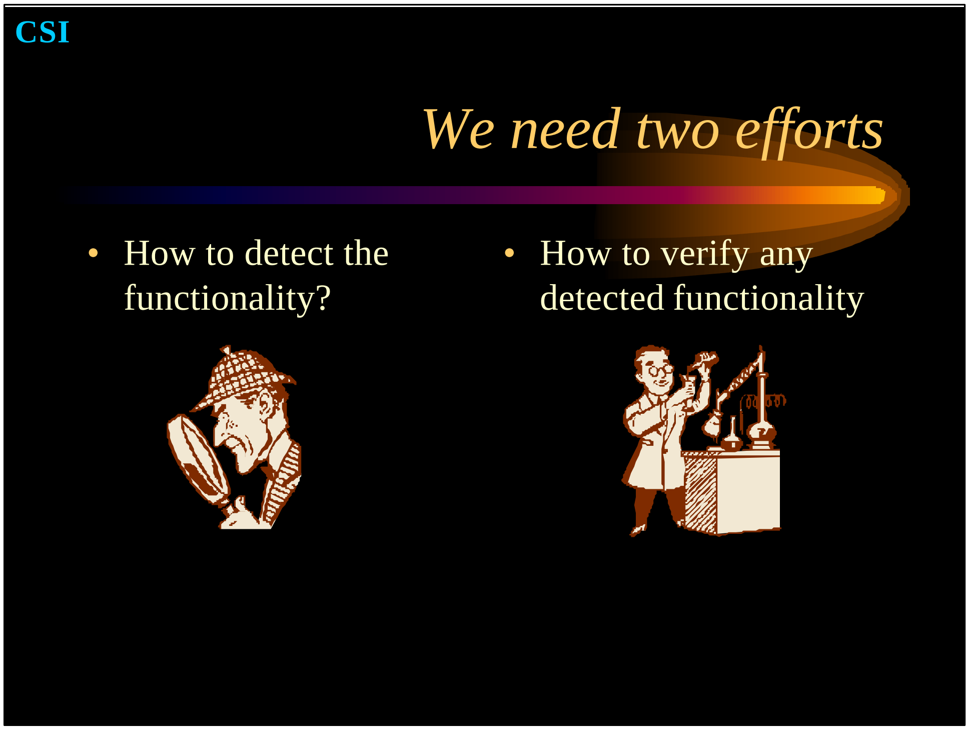
The above code fragment represents a test on a logical operation. The logical operation involves a division before the test is made. The compiler decides that before any division is performed it will check for a zero dividend and then fix it up so that the division will occur but wont result in an overflow. This is done in the Call DIV0 routine. There is no evidence of this from the source code. This may even be a built in compiler function that doesn't require any compile options (unlikely). This brings up two issues. One is how to detect that this functionality exists and second how to ensure that it behaves correctly and doesn't create anomalous behavior. Note that the If-Then-Else assembly code is incorrect. The idea behind sections 4.4.2 and 6.4.4.2 in DO-178B was to detect the former and not the latter. In all but extremely rare cases the normal verification activity will pick up the latter condition.



CSI

What does 178B Really Say

- 4.4.2 (b) - To implement certain features, compilers for some languages may produce object code that is not directly traceable to the source code, for example, initialization, built-in error detection or exception handling (subparagraph 6.4.4.2, item b). The software planning process should provide a means to detect this object code and to ensure verification coverage and define the means in the appropriate plan.
- 6.4.4.2 (b) - The structural coverage analysis may be performed on the Source Code, unless the software level is A and the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences. A compiler-generated array-bound check in the object code is an example of object code that is not directly traceable to the Source Code



CSI

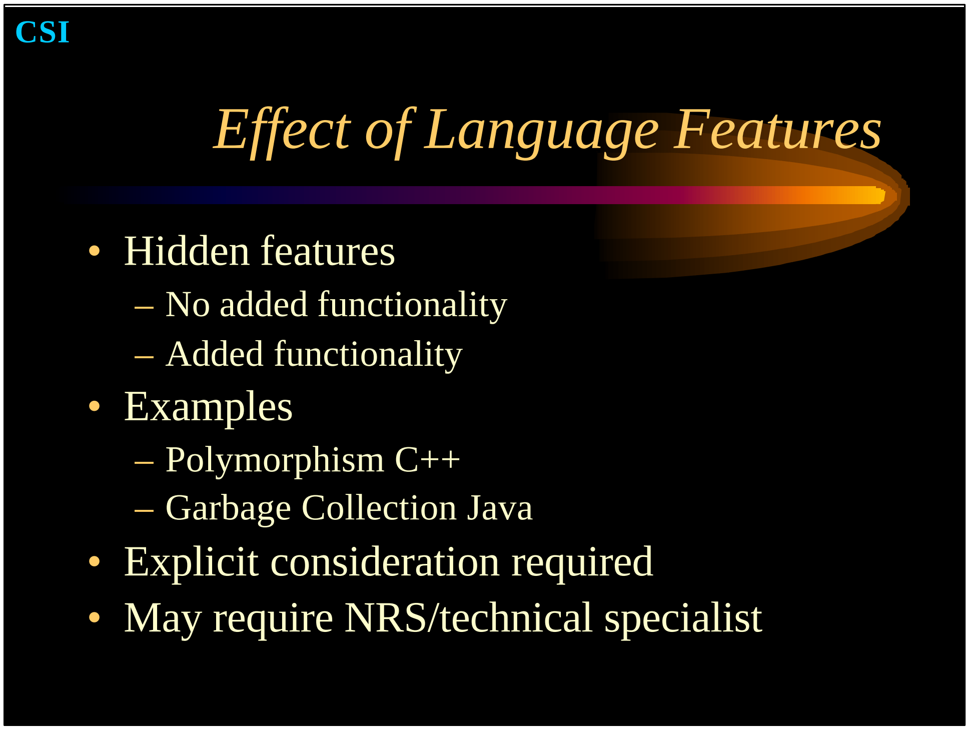
We need two efforts

- How to detect the functionality?
- How to verify any detected functionality

This all boils down to the following when decoding the DNA in sections 4.4.2 and 6.4.4.2 .

1. We must make it clear that these sections do not require any activity to ensure that generated code sequences are correct. This is accomplished under other objectives.

2. We need to ensure that the giudance in this section results in a requirement to perform structural coverage at the object code level.



CSI

Effect of Language Features

- Hidden features
  - No added functionality
  - Added functionality
- Examples
  - Polymorphism C++
  - Garbage Collection Java
- Explicit consideration required
- May require NRS/technical specialist

A number languages have features that complicate the ability to detect or determine if there is added functionality. In some cases the visible language is just a meta langauge and quite different than what is finally installed in the machine. Polymorphism from C++ is a good example of this type of issue. What appears to be a simple statement in the C++ code is actually part of a case statement which established which executable is executed. There may be cases not executed during the normal evaluation or there may be compiler added robustness added.

There should be explicit determination of the impact of any language being used.